# Synthesizing Entity Matching Rules by Examples

Rohit Singh♠ Vamsi Meduri♢ Ahmed Elmagarmid♣ Samuel Madden♠
Paolo Papotti♡ Jorge-Arnulfo Quiané-Ruiz♣ Armando Solar-Lezama♠ Nan Tang♣
♠CSAIL, MIT, USA ♢Arizona State University, USA
♣Qatar Computing Research Institute, HBKU, Qatar ♡EURECOM, France
{rohitsingh, madden, asolar}@csail.mit.edu, vmeduri@asu.edu,
{aelmagarmid, jquianeruiz, ntang}@hbku.edu.qa, papotti@eurecom.fr

## ABSTRACT

Entity matching (EM) is a critical part of data integration. We study how to *synthesize entity matching rules* from positive-negative matching examples. The core of our solution is *program synthesis*, a powerful tool to automatically generate rules (or programs) that satisfy a given high-level specification, via a predefined grammar. This grammar describes a *General Boolean Formula* (**GBF**) that can include arbitrary attribute matching predicates combined by conjunctions ($\bigwedge$), disjunctions ($\bigvee$) and negations ($\neg$), and is expressive enough to model EM problems, from capturing arbitrary attribute combinations to handling missing attribute values. The rules in the form of **GBF** are more concise than traditional EM rules represented in Disjunctive Normal Form (**DNF**), and consequently more interpretable than decision trees and other machine learning algorithms that output deep trees with many branches. We present a new synthesis algorithm that, given only positive-negative examples as input, synthesizes EM rules that are effective over the entire dataset. Extensive experiments show that we outperform other interpretable rules (e.g., decision trees with low depth) in effectiveness, and are comparable with non-interpretable tools (e.g., decision trees with high depth and SVM).

## 1. INTRODUCTION

Entity matching (EM), where a system or user finds records that refer to the same real-world object, is a fundamental problem of data integration [13] and cleaning [5].

There is a key tension in EM solutions: on one side, machine learning (ML)-based solutions are often preferred, as they typically offer higher effectiveness. On the other side, hand-crafted rules are also desirable, because their logical structure makes them *interpretable* by humans. Interpretable rules enable interactive debugging of the results [30], maintenance [10], explicit definition of domain knowledge [8], and optimization at execution time [17].

Unfortunately, interpretability is not supported by systems that use ML methods – such as SVMs [7] or fuzzy matching [18] – because their models consist of weights and functional parameters that are hard to interpret even for technical users. In contrast, rule-based systems [17] offer better *interpretability*, particularly when the rules can be constrained to be simple with relatively few clauses.

For this reason, despite the recent effort for new ML approaches, several data-centric systems privilege rule-based approaches in tasks similar to EM. A survey over 54 Information Extraction vendors shows that 67% of the tools are rule-based, only 17% ML-based, and 16% a hybrid of the two [10]. In Product Classification at Walmart [8], rules are adopted because they allow domain analysts to improve the system, without the involvement of a CS developer to revise a statistical model. For these reasons, developing interpretable ML models is an important challenge in the ML community [23, 26]. However, since hand-writing EM rules is extremely time consuming and error-prone, a key question is whether we can automatically generate interpretable EM rules, by learning from positive-negative matching examples.

We present a system to learn EM rules that (1) matches the performance of ML methods, and (2) produces concise rules [34], as we consider a model more *interpretable* than another if it consists of fewer logical predicates or atoms. Our approach is to use *program synthesis* [37] (PS), in which a program (set of rules) is generated by using positive-negative examples as constraints that guide the synthesizer towards rules that match the examples.

**Example 1:** Consider two tables of famous people that are shown in Figure 1. Dataset $D_1$ is an instance of schema $R$ (name, address, email, nation, gender) and $D_2$ is of schema $S$ (name, apt, email, country, sex). The EM problem is to find tuples in $D_1$ and $D_2$ that refer to the same person.

*Off-the-shelf* schema matching tools [14, 33] may decide that name, address, email, nation, gender in table $R$ map to name, apt, email, country, sex in table $S$, respectively.

Given a tuple $r \in D_1$ and a tuple $s \in D_2$, a straightforward EM rule is that *the value pairs from all aligned attributes should match* such as:

$$\varphi_1\colon r[\text{name}] \approx_1 s[\text{name}] \bigwedge r[\text{address}] \approx_2 s[\text{apt}] \bigwedge r[\text{email}] = s[\text{email}] \bigwedge r[\text{nation}] = s[\text{country}] \bigwedge r[\text{gender}] = s[\text{sex}]$$

Here, $\approx_1$ and $\approx_2$ are different domain-specific similarity functions. Rule $\varphi_1$ says that a tuple $r \in D_1$ and a tuple $s \in D_2$ refer to the same person (i.e., a match), if they have *similar* or *equivalent* values on all aligned attributes. □

However, in practice, the rule $\varphi_1$ above may result in very low recall, since real-world data may contain multiple issues such as misspellings (e.g., $s_3$[name]), different formats (e.g.,


*Proceedings of the VLDB Endowment,* Vol. 10, No. 11

| | name | address | email | nation | gender |
|---|---|---|---|---|---|
| $r_1$ | Catherine Zeta-Jones | 9601 Wilshire Blvd., Beverly Hills, CA 90210-5213 | c.jones@gmail.com | Wales | F |
| $r_2$ | C. Zeta-Jones | 3rd Floor, Beverly Hills, CA 90210 | c.jones@gmail.com | US | F |
| $r_3$ | Michael Jordan | 676 North Michigan Avenue, Suite 293, Chicago | | US | M |
| $r_4$ | Bob Dylan | 1230 Avenue of the Americas, NY 10020 | | US | M |

(a) $D_1$: **an instance of schema** $R$

| | name | apt | email | country | sex |
|---|---|---|---|---|---|
| $s_1$ | Catherine Zeta-Jones | 9601 Wilshire, 3rd Floor, Beverly Hills, CA 90210 | c.jones@gmail.com | Wales | F |
| $s_2$ | B. Dylan | 1230 Avenue of the Americas, NY 10020 | bob.dylan@gmail.com | US | M |
| $s_3$ | Micheal Jordan | 427 Evans Hall #3860, Berkeley, CA 94720 | jordan@cs.berkeley.edu | US | M |

(b) $D_2$: **An instance of the schema** $S$

**Figure 1: Sample tables for persons**

$r_2$[name] and $s_1$[name]), and missing values (e.g., $r_3$[email] and $r_4$[email]). Naturally, a robust solution is to have a set of rules that collectively cover different cases.

**Example 2:** For example, we may have two rules as below.

$\varphi_2$: $r$[name] $\approx_1$ $s$[name] $\bigwedge$ $r$[address] $\approx_2$ $s$[apt] $\bigwedge$ $r$[nation] $=$ $s$[country] $\bigwedge$ $r$[gender] $=$ $s$[sex];
$\varphi_3$: $r$[name] $\approx_3$ $s$[name] $\bigwedge$ $r$[email] $=$ $s$[email]

Typically, these kinds of rules are specified as disjuncts, e.g., $\varphi_2 \bigvee \varphi_3$, which indicates that a tuple $r \in D_1$ and a tuple $s \in D_2$ match, if either $\varphi_2$ or $\varphi_3$ holds. However, a more natural way, from a user perspective, is to specify the rule in a logical flow. For instance, when handling Null values, the following representation may be more user-friendly:

$\varphi_4$: **if** ($r$[email] $\neq$ Null $\bigwedge$ $s$[email] $\neq$ Null)
**then** $r$[name] $\approx_1$ $s$[name] $\bigwedge$ $r$[email] $=$ $s$[email]
**else** $r$[name] $\approx_3$ $s$[name] $\bigwedge$ $r$[address] $\approx_2$ $s$[apt] $\bigwedge$ $r$[nation] $=$ $s$[country] $\bigwedge$ $r$[gender] $=$ $s$[sex]

These `if-then-else` rules provide a more flexible way to model matching rules. □

**Challenges.** There are two key challenges in automatically discovering good EM rules from examples.

*(1) Interpretability vs. Effectiveness.* While interpretability is crucial in many domains, it also might sacrifice the effectiveness of the system. In real-world applications, it is often difficult or impossible to find matching tuples by considering only a few attribute combinations. A solution to the EM rule mining problem should keep rules simple and concise but still achieve satisfactory effectiveness.

*(2) Large Search Space.* Consider two relations with $n$ aligned attributes. There are $m = 2^n$ possible combinations of attributes. If we constrain ourselves to EM rules that consist of arbitrary selections of these attribute combinations represented in **DNF** (disjunctive normal form), this results in a search space of $\Sigma_{i=1}^{m}\binom{m}{i} = 2^m - 1 = 2^{2^n} - 1$. For instance, the search space is $2^{2^5} - 1$ (4 billion combinations) for Example 2, which contains only 5 attributes! Moreover, we need to consider the possible similarity function and threshold for each attribute.

**Contributions.** We propose a new EM rule synthesis engine to generate rules that are both concise and effective (Challenge 1). The core of our approach is an algorithm based on *Program Synthesis* [37] (PS), in the *Syntax-Guided Synthesis* (SyGuS) framework [6]. Given a predefined grammar for EM rules (hardwired into our system for all datasets), PS is optimized to explore the massive space of possible rules and find rules that satisfy the provided examples (Challenge 2). Unfortunately, existing SyGuS solvers are designed to find solutions that satisfy all examples, and have a difficult time reasoning about complex numerical functions such as the similarity functions used by EM rules. To cope with these challenges, we devise a novel algorithm RuleSynth, which adopts the idea of Counter-Example Guided Inductive Synthesis (CEGIS) [38] to perform synthesis from small sets of examples, and is inspired by Random Sample Consensus (RANSAC) [20] to avoid examples that may make the algorithm under-perform, and we combine it with a new approach for combining special purpose search procedures with a general purpose constraint-based synthesizer.

We summarize our contributions as follows:

(1) We define the problem of synthesizing EM rules from positive-negative examples. In particular, we use *General Boolean Formulas* (**GBF**) to represent EM rules, and define an optimization problem to find a *good* EM rule (Section 2).

(2) We show how to solve this optimization problem using the SyGuS framework (Section 3). We also develop a new algorithm, built on top of an open-source SyGuS engine named Sketch [37], to synthesize EM rules from positive-negative examples (Section 4).

(3) We describe optimizations in our algorithm to avoid over-fitting, to eliminate biased samples, and to compute the composition of multiple rules (Section 5).

(4) We experimentally verify that our system significantly outperforms other interpretable models (i.e., decision trees with low depth, SIFI [40]) in terms of matching accuracy, but our rules have much fewer clauses. It is also comparable with other uninterpretable models, e.g., decision trees with large depth and SVM, on accuracy (Section 6).

# 2. PROBLEM OVERVIEW

## 2.1 Notation

Let $R[\mathsf{A}_1, \mathsf{A}_2, \ldots, \mathsf{A}_n]$ and $S[\mathsf{A}'_1, \mathsf{A}'_2, \ldots, \mathsf{A}'_n]$ be two relations with corresponding sets of $n$ aligned attributes $\mathsf{A}_i$ and $\mathsf{A}'_i$ ($i \in [1, n]$). We assume that the attributes between two relations have been aligned and provided as an input. Note that our approach naturally applies to one relation.

Let $r, s$ be records in $R, S$ and $r[\mathsf{A}_i], s[\mathsf{A}'_i]$ be the values of the attribute $\mathsf{A}_i, \mathsf{A}'_i$ in records $r, s$, respectively.

A *similarity function* $f(r[\mathsf{A}_i], s[\mathsf{A}'_i])$ computes a similarity score in the real interval $[0, 1]$, e.g., edit distance and Jaccard similarity. A bigger score means that $r[\mathsf{A}_i]$ and $s[\mathsf{A}'_i]$ have a higher similarity.

**Attribute-Matching Rules.** An *attribute-matching rule* is a triple $\approx(i, f, \theta)$ representing a Boolean function with value $f(r[\mathsf{A}_i], s[\mathsf{A}'_i]) \geqslant \theta$, where $i \in [1, n]$ is an index, $f$ is a similarity function and $\theta \in [0, 1]$ is a threshold value. Attribute-matching rule $\approx(i, f, \theta)$ evaluating to *true* means

that $r[\mathsf{A_i}]$ *matches* $s[\mathsf{A'_i}]$ relative to the specific similarity function $f$ and threshold $\theta$.

We write $r[\mathsf{A_i}] \approx_{(f,\theta)} s[\mathsf{A'_i}]$ as an attribute-matching rule for a similarity function $f$ and threshold $\theta$. We will simply write $r[\mathsf{A_i}] \approx s[\mathsf{A'_i}]$ when it is clear from the context.

**Record-Matching Rules.** A *record-matching rule* is a conjunction of a set of attribute-matching rules on different attributes. Intuitively, two records $r$ and $s$ *match* iff all attribute-matching rules in the set evaluate to *true*.

**Disjunctive Matching Rule.** A *disjunctive matching rule* is a disjunction of a set of record-matching rules. Records $r$ and $s$ are matched by this rule *iff* they are matched by at least one of this rule's record-matching rules.

Indeed, a *disjunctive matching rule* can be seen as a formula in Disjunctive Normal Form ($\mathbf{DNF_{EM}}$) over *attribute-matching rules* as: $\bigvee_{p=1}^{P} \left( \bigwedge_{q=1}^{Q_p} \approx(i_{(p,q)}, f_{(p,q)}, \theta_{(p,q)}) \right)$

Note that we use $\mathbf{DNF_{EM}}$ to refer to the EM rules that are traditionally used for the EM problem, which is more restricted than the general **DNF** in Boolean logic. There are two main shortcomings of using $\mathbf{DNF_{EM}}$ rules:

(1) [Not Concise.] A $\mathbf{DNF_{EM}}$ $(u_1 \bigwedge v_1) \bigvee (u_1 \bigwedge v_2) \bigvee (u_2 \bigwedge v_1) \bigvee (u_2 \bigwedge v_2)$ is equivalent to a much more concise formula $(u_1 \bigvee u_2) \bigwedge (v_1 \bigvee v_2)$.

(2) [Expressive Power.] A $\mathbf{DNF_{EM}}$ rule without negations cannot express the logic "if $(u)$ then $(v)$ else $(w)$", which can be modeled using a formula such as $(u \bigwedge v) \bigvee (\neg u \bigwedge w)$. Traditionally, negations are not used in positive EM rules.

Hence, a more natural way than $\mathbf{DNF_{EM}}$ to define EM rules is to use general boolean formulas, as defined below.

**Boolean Formula Matching Rule.** A *Boolean formula matching rule* is an arbitrary *Boolean formula* over attribute-matching rules as its variables and conjunction ($\bigwedge$), disjunction ($\bigvee$) and negation ($\neg$) as allowed operations.

We formulate a *Boolean formula matching rule* as a *General Boolean Formula* (**GBF**).

**Example 3:** Consider Example 2. Let the similarity function for matching attributes name in $R$ and name in $S$ (resp. address in $R$ and apt in $S$) be Levenshtein (resp. Jaccard), with threshold 0.8 (resp. 0.7).

[Attribute-matching rule.] $r[\mathsf{name}] \approx s[\mathsf{name}]$ can be formally represented as $\approx(1, \texttt{Levenshtein}, 0.8)$, where the number 1 is the positional index for the $1^{st}$ pair of aligned attributes, i.e., attributes (name, name) for relations ($R$, $S$).

[Record-matching rule.] $\varphi_2$ and $\varphi_3$ can be formalized as:

$\varphi_2 : \approx(1, \texttt{Levenshtein}, 0.8) \bigwedge \approx(2, \texttt{Jaccard}, 0.7) \bigwedge =(4, \texttt{Equal}, 1.0) \bigwedge =(5, \texttt{Equal}, 1.0)$

$\varphi_3 : \approx(1, \texttt{Levenshtein}, 0.8) \bigwedge =(3, \texttt{Equal}, 1.0)$

[Disjunctive matching rule.] A disjunctive matching rule for $\varphi_2$ and $\varphi_3$ is the disjunction of the above two record-matching rules, $\varphi_2 \bigvee \varphi_3$.

[Boolean formula matching rule.] Consider a custom *similarity* function noNulls that returns 1.0 when the values of the corresponding attributes are both not null and 0.0 otherwise. Using this function, we can formalize $\varphi_4$ as:

$\varphi_4$: **if** $(\approx(1, \texttt{noNulls}, 1.0))$ **then** $\varphi_2$ **else** $\varphi_3$ □

There are two reasons why we propose to synthesize **GBF** rules instead of $\mathbf{DNF_{EM}}$ rules. (1) **GBF** can concisely represent a $\mathbf{DNF_{EM}}$ and increase its expressibility thereby enhancing the readability. (2) Traditionally used EM rules in the $\mathbf{DNF_{EM}}$ [40] form require each attribute to show up with the same similarity function and threshold everywhere in the $\mathbf{DNF_{EM}}$, with the main purpose of reducing the search space of their solution. In our **GBF** rules, we allow one attribute to have different similarity functions in the *Boolean formula*, since values in the same column are not always homogeneous, and we need different similarity functions to capture different matching cases. Consider for instance the attribute name. In rule $\varphi_2$, the similarity function used is Levenshtein with threshold 0.8. A variant $\varphi'_3$ of $\varphi_3$ could use Jaccard similarity with threshold 0.6 for name.

## 2.2 Problem Statement

We want to generate an *optimal* general Boolean formula (**GBF**) without user involvement in providing structure for the **GBF**. To evaluate the quality of a **GBF**, we assume that the user provides a set of examples, denoted by $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$, where $\mathbf{M}$ are positive examples, i.e., pairs of records that represent the same entity, and $\mathbf{D}$ are negative examples, i.e., pairs of records that represent different entities.

**Optimality Metric.** Consider a **GBF** $\Phi$ and positive and negative examples $\mathbf{M}$ and $\mathbf{D}$. We define a metric $\mu(\Phi, \mathbf{M}, \mathbf{D})$ returning a real number in $[0, 1]$ that quantifies the *goodness* of $\Phi$. The larger the value of $\mu$, the better is $\Phi$.

Let $\mathbf{M}_\Phi \subset \mathbf{E}$ be the set of all examples $(r', s')$ such that $r'$ and $s'$ are *matched* by $\Phi$. Alternative candidates for optimality metric $\mu$ are:

$$\mu_{\text{precision}} = \frac{|\mathbf{M}_\Phi \cap \mathbf{M}|}{|\mathbf{M}_\Phi \cap \mathbf{M}| + |\mathbf{M}_\Phi \cap \mathbf{D}|}$$

$$\mu_{\text{recall}} = \frac{|\mathbf{M}_\Phi \cap \mathbf{M}|}{|\mathbf{M}|}$$

$$\mu_{\text{F-measure}} = \frac{2 \cdot \mu_{\text{precision}} \cdot \mu_{\text{recall}}}{\mu_{\text{precision}} + \mu_{\text{recall}}}$$

**Problem Statement (EM-GBF).** Given two relations $R$ and $S$, the aligned attributes between $R$ and $S$, sets $\mathbf{M}$ and $\mathbf{D}$ of positive and negative examples, a library of similarity functions $\mathcal{F}$, and an optimality metric $\mu$, the **EM-GBF** problem is to discover a **GBF** $\Phi$ that maximizes $\mu$.

# 3. SYNTHESIS OVERVIEW

In this section, we introduce program synthesis (PS), and describe how to formulate the **EM-GBF** problem as a PS problem. The basic idea of PS is to search for a complete program that satisfies a set of input-output examples, given a grammar that specifies what is valid in the programming language of choice. In the context of EM, the grammar represents the space of **GBF**s, and the problem is to search for a **GBF** that satisfies as many examples as possible.

Our system assumes that the two input relations' attributes have been aligned. Afterwards, to solve the synthesis problem our system must find the best structure of the **GBF** along with the best combinations of attributes, similarity functions, and thresholds for each attribute matching rule in the **GBF**.

Internally, we use the formalism of partial programs to frame the synthesis problem. In these partial programs, *holes* are used to represent the unknown aspects of the program, and the space of all possible code fragments that can fill those holes is given by a *grammar*. The partial program also includes *behavioral constraints*, which in the case of our problem require the synthesized program to match the examples (Section 3.1). This style of synthesis based on partial

programs, behavioral constraints, and grammars of possible code fragments is known as Syntax Guided Synthesis (SyGuS) and it has recently received significant attention by the formal methods community [6]. We give the modeling of the **EM-GBF** problem as a SyGuS problem and show how the appropriate grammar can be represented as a partial program in an open-source SyGuS solver, namely SKETCH [37] (Section 3.2). Note, however, that not all examples can be satisfied in practice, we also define a version of the SyGuS problem as an optimization problem (Section 3.3).

## 3.1 Partial Programs and Grammars

**Partial Programs.** A *partial program* represents a space of possible programs by giving the synthesizer explicit choices about what code fragments to use in different places. A simple partial program in SKETCH is shown below:

```
void tester(bit x, bit y) {
    bit t = boolExp(x, y);
    if (x) assert t ==~y;
    if (y) assert t == ~x; }
bit boolExp(bit x, bit y){
     return {| ((x | ~x) || (y |~y)) |}; }
```

The partial program (called a sketch) gives the synthesizer choices separated by |, about whether or not to negate $x$ and $y$ before or-ing them together in order to satisfy the assertions in the tester. The language also supports *unknown constants* indicated with "??" which the solver replaces with concrete values (as employed in Example 4 below).

**Grammar.** SyGuS problems are also represented abstractly as a *grammar* $G$ representing a set of expressions, together with a constraint $C$ on the behavior of the desired expression. A *grammar* $G$ is a set of recursive rewriting rules (or *productions*) used to generate expressions over a set of *terminal symbols* and *non-terminal (recursive) symbols*. The productions provide a way to *derive* expressions from a designated *initial symbol* by applying the productions one after another. An example SyGuS problem with a grammar and an associated constraint is given below:

$$\begin{array}{ll} \textsf{grammar} & expr \rightarrow expr \vee expr \ (\textbf{bound} : B) \\ & expr \rightarrow x \mid y \mid \neg x \mid \neg y \\ \textsf{constraint} & (x \Rightarrow \neg y = expr) \wedge (y \Rightarrow \neg x = expr) \end{array}$$

The above grammar has a non-terminal symbol (also the initial symbol) *expr* that represents a disjunction of variables $x$, $y$ or their negations. Unlike the sketch above, the space of expressions is unbounded, except for a parameter $B$ that bounds the number of times a rule can be used. SKETCH also supports recursive definitions of program spaces that are equivalent to the grammar above; for the rest of the paper we alternate between showing more abstract descriptions of a space of expressions as a grammar, and showing more concrete SKETCH syntax when necessary.

## 3.2 SyGuS Components for EM-GBF

We are ready to give the grammar and constraints to formulate the **EM-GBF** problem. It is important to emphasize that these partial programs are built into the tool; the encoding of the grammar as sketches is invisible to the user.

**Grammar for EM-GBF.** In order to formulate the **EM-GBF** problem in the SyGuS framework, we use a generic Boolean formula grammar ($G_{\mathbf{GBF}}$) defined below:

$$\begin{array}{ll} \textsf{grammar} & G_{\textsf{attribute}} \rightarrow r[\mathsf{A_i}] \approx_{(f,\theta)} s[\mathsf{A'_i}] \\ & i \in [1,n]; f \in \mathcal{F}; \theta \in [0,1] \\ \textsf{grammar} & G_{\mathbf{GBF}} \rightarrow G_{\textsf{attribute}} \ (\textbf{bound} : N_a) \\ & \left.\begin{array}{l} G_{\mathbf{GBF}} \rightarrow \neg G_{\mathbf{GBF}} \\ G_{\mathbf{GBF}} \rightarrow G_{\mathbf{GBF}} \wedge G_{\mathbf{GBF}} \\ G_{\mathbf{GBF}} \rightarrow G_{\mathbf{GBF}} \vee G_{\mathbf{GBF}} \end{array}\right\} (\textbf{bound} : N_d) \end{array}$$

The grammars $G_{\textsf{attribute}}$ and $G_{\mathbf{GBF}}$ represent an attribute-matching rule and a Boolean formula matching rule (**GBF**), respectively. Note that the search space represented by the above grammars is infinite because there are infinitely many real values for $\theta \in [0,1]$. We tackle this by introducing a custom synthesis procedure (Section 4.2). The bounds $N_a$ and $N_d$ make the search space for the Boolean formula finite by bounding the number of attribute-matching rules ($G_{attribute}$) in $G_{\mathbf{GBF}}$ and the number of recursive productions being used, i.e., depth of the expansion of the grammar, respectively.

**Constraints for EM-GBF.** A candidate selected from the grammar $G_{\mathbf{GBF}}$ can be interpreted as a Boolean formula. Given both positive (**M**) and negative (**D**) examples, the SyGuS constraints are specified as the evaluation of this **GBF** on the provided examples being consistent:

$$\begin{array}{ll} \textsf{constraint} & G_{\mathbf{GBF}}(r_m, s_m) = \mathit{true} \ \forall \ (r_m, s_m) \in \mathbf{M} \\ \textsf{constraint} & G_{\mathbf{GBF}}(r_d, s_d) = \mathit{false} \ \forall \ (r_d, s_d) \in \mathbf{D} \end{array}$$

**Partial Programs for EM-GBF.** Now let's showcase the partial programs used for the **EM-GBF** problem.

**Example 4:** Consider the two tables discussed in Example 1. The partial program that represents a Boolean formula matching rule (**GBF**) with $N_a$ attribute-matching rules and $N_d$ depth of grammar expansion is listed below.

```
grammar bool attributeRule(int e){ // e = Example Id
    int i = ??; // Attribute Id
    assert (1 <= i && i <= 5);
    int f = ??; // Similarity Fn Id
    assert (1 <= f && f <= 29);
    double θ = customSynth(i,f);
    return (evalSimFn(e,i,f) >= θ);
}

@depth(N_d)
grammar bool gbfRule(int e, int &A){
    if (??){ A++; return attributeRule(e); }
    else if (??) return ! (gbfRule(e,A));
    else if (??) return gbfRule(e,A) && gbfRule(e,A);
    else return gbfRule(e,A) || gbfRule(e,A);
}

bool matchingRule(int e, int N_a){
  int A=0;
  bool b = gbfRule(e,A);
  assert (A<=N_a);
  return b;
}

constraint void examples(int N_a){
  //Example Id 1 is a positive example
  assert(macthingRule(1,N_a) == true);
  //Example Id 2 is a negative example
  assert(macthingRule(2,N_a) == false);
}
```

In the code above, some functions are annotated with being a `grammar` or a `constraint`. For example, `attributeRule` is a `grammar` function. Since there are 5 aligned attributes, we assert that the values taken by $i$ lie between 1 and 5. Similarly, the candidate space of 29 similarity functions is asserted accordingly. The values for threshold $\theta$ are chosen using a custom synthesis procedure. The

function `evalSimFn` symbolically represents the evaluation of function $f$ on attribute $i$ of the records from example $e$ (see more details in Section 4.2). Also, `gbfRule` is a `grammar` function with function `attributeRule` being inlined at most $N_a$ times (enforced by a variable $A$ passed by reference) and multiple recursive calls to itself to specify the possible expansion of the grammar. The expansion is bounded by a depth $N_d$ passed as a parameter in the "`@`" annotation. Note that, in SKETCH, each `grammar` function is completely inlined up to the specified depth as a parameter. This results into the *holes* ("??"s) occurring multiple times as well. Each hole inside the `if`'s represents a possible *true* or *false* value.

The `examples` function is a `constraint` that represents the requirement that the resulting rule should work for the positive and negative examples.

The SKETCH synthesizer will fill all the holes in the above partial program to synthesize a *complete program*, with a function `matchingRule` that represents a Boolean formula (**GBF**) for entity matching. □

**Wrap Up.** Next we put together the sample grammars and constraints to show how to obtain a **GBF**.

**Example 5:** Consider the example in Figure 1. A specific grammar $G^5_{\mathbf{GBF}}$ for representing a Boolean formula matching rule (**GBF**) in this scenario is obtained by using the following in the above definition of the grammar $G_{\mathbf{GBF}}$:

– let $n = 5$ (number of aligned attributes),

– let $\mathcal{F} = \{\texttt{Equal}, \texttt{Levenshtein}, \texttt{Jaccard}\}$,

– let examples be: matching $\mathbf{M} = \{(r_1, s_1), (r_2, s_1)\}$ and non-matching $\mathbf{D} = \{(r_1, s_2)\}$

Our system gives the synthesizer a table representing the evaluations of each similarity function $f \in \mathcal{F}$ on each attribute $i \in [1, n]$ of every provided example $(r, s) \in \mathbf{E}$ (the function `evalSimFn` in the sketch). The **GBF** $\varphi_2 \vee \varphi_3$ from Example 2 can now be obtained as candidate **GBF** from this grammar $G^5_{\mathbf{GBF}}$. □

## 3.3 Optimization SyGuS Problem

We are ready to model the **EM-GBF** problem by extending the *Syntax-Guided Synthesis (SyGuS) framework* [6]. We consider two versions of the problem: the *exact problem* and the *optimization problem*. The former corresponds to finding a **GBF** that satisfies all constraints from examples. Unfortunately, such a perfect **GBF** oftentimes does not exist in practice because examples may have errors or the grammar may not be expressive enough to correctly classify all examples. The latter relaxes the condition by discovering a **GBF** of partial satisfaction of constraints based on an optimality metric $\mu$. The **EM-GBF** problem is equivalent to the optimization version of the SyGuS problem defined below.

**Optimization SyGuS Problem (EM-Synth).** Given a grammar and constraints from positive-negative examples, the *optimization SyGuS problem* is to find a candidate **GBF** in the grammar that satisfies a subset of the constraints that maximizes the given optimality metric $\mu$.

As will be seen shortly, although exact SyGuS cannot solve the studied **EM-Synth** problem, it can still be used as a building block in our algorithm (Section 4).

# 4. SYNTHESIS ALGORITHMS

Existing SyGuS solvers are designed to solve the *exact SyGuS problem*, not the *optimization SyGuS problem* (**EM-Synth**) that would discover a **GBF** that maximizes a given optimization metric. In this section, we start by giving a naïve solution (Section 4.1) to solve **EM-Synth**. We then present our novel RULESYNTH algorithm (Section 4.2).

## 4.1 A Naïve Solution & Its Limitations

In fact, given a set **E** of examples, grammars, and constraints, the **GBF** that satisfies all constraints from all examples may not exist. Hence, we shift our goal to find a **GBF** that satisfies all constraints for a *subset* of examples.

**A Naïve Solution.** Informally, a simple approach would be to choose multiple, random, subsets **S** from all examples **E**, and invoke the SKETCH SyGuS solver on each subset in **S**. The solver will only succeed on some of these subsets, but for those that it succeeds on, we can take the best performing **GBF** based on the optimality metric $\mu$ evaluated on all examples in **E**.

**Limitations.** The naïve solution has three limitations.

(i) We must choose subsets of **E** in a way that allows us to synthesize a **GBF** with good coverage of the example sets.

(ii) We have to avoid examples that do not lead to a *good* solution, i.e., *sub-optimal* examples that are not matched correctly by any matching rule with high $\mu$ value.

(iii) We have to reason about numerical similarity functions and thresholds in a symbolic solver like SKETCH, but such reasoning is not supported by existing solvers.

## 4.2 A Novel Solution (RULESYNTH)

Below we introduce our new algorithm. For Limitation (i), we use ideas from the Counter-Example Guided Inductive Synthesis (CEGIS) [38] to perform synthesis from a few examples. For Limitation (ii), we are inspired by the Random Sample Consensus (RANSAC) [20] to avoid sub-optimal examples. For Limitation (iii), we add a custom synthesizer for finding a numerical threshold within the symbolic solver.

Note that, the synthesis approach used by SKETCH relies on having a complete symbolic representation of all the building blocks of the program, but when the building blocks are complex numerical functions, as our similarity functions are, the process can become very inefficient. In this paper, we pioneer a new technique that allows the general purpose solver to collaborate with a special purpose search procedure that can reason about the similarity functions and their numerical thresholds. This is conceptually challenging because of the different approaches that the two solvers use to represent the search space. Our novel technique allows the individual synthesis instances to be solved in seconds instead of the hours they take on using only the SKETCH general purpose synthesizer.

**Algorithm**. The algorithm, referred to as RULESYNTH, is presented in Algorithm 1 with an overview in Figure 2.

It has two loops. The outer (RANSAC) loop (lines 3-19) picks random samples to bootstrap the synthesis algorithm (step 1 in Figure 2). In each iteration, given a sample (line 5), it starts with the **Synth** routine (line 6). It then invokes the inner (CEGIS) loop (lines 7-18). In each iteration, it first synthesizes a **GBF** (line 8). If it cannot find a satisfiable **GBF**, it will restart (lines 9-10 and step 4); otherwise, it will **Verify** to find counter-examples (line 11 and step 3). Either there is no counter-example so the process will terminate (lines 12-13 and step(5)), or a randomly

**Algorithm 1:** Synthesis Algorithm for **EM-Synth**

**input :** $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$ : Set of examples
$G_{\mathbf{GBF}}(N_a, N_d)$ : Bounded **GBF** grammar
$\mathcal{F}$ : Library of Similarity Functions
$\mu$ : Optimality metric
$K_{\text{RANSAC}}$ : Bound on RANSAC restarts
$K_{\text{CEGIS}}$ : Bound on CEGIS iterations
**output:** $\Phi$* : A **GBF** from $G_{\mathbf{GBF}}(N_a, N_d)$ maximizing $\mu$

```
1  r ← 0
2  Φ* ← true
3  while r < K_RANSAC do                                  // RANSAC loop
4      i ← 0
5      e_0 ← sample (E)
6      E_SYN ← List(e_0)
7      while i < K_CEGIS do                               // CEGIS loop
8          Φ_i ← Synth (G_GBF(N_a, N_d), E_SYN, F)
9          if Φ_i = null then                             // Unsatisfiable Synth
10             break                                      // restart CEGIS
11         Ē_{Φ_i} ← Verify (Φ_i, E)                      // Counter-examples
12         if Ē_{Φ_i} = ∅ then
13             return Φ_i
14         else
15             e_{i+1} ← sample (Ē_{Φ_i})
16             E_SYN ← E_SYN.append(e_{i+1})
17         Φ* = arg max_{Φ∈{Φ*,Φ_i}} μ(Φ, M, D)
18         i ← i + 1
19     r ← r + 1
20 return Φ*
```

selected counter-example will be added to be considered in the next CEGIS iteration (lines 14-16 and step 3). The current best **GBF** will be re-calculated (line 17). Finally, the algorithm will return a **GBF** (line 20).

We explain different parts of the algorithm below.

**Customized Synth Routine:** We begin with the core **Synth** routine (line 8) that solves the exact SyGuS problem, i.e., it searches for a candidate **GBF** from the bounded grammar $G_{\mathbf{GBF}}(N_a, N_d)$ that satisfies all the constraints arising from examples in $\mathbf{E}_{\text{SYN}}$. SKETCH solver works by analyzing every part of the grammar and constraints symbolically, and, reducing the search problem to a Boolean satisfiability (SAT) problem. Using SKETCH directly for this problem is impractical because it involves reasoning about complicated numerical functions. For solving this problem with SKETCH, we pioneer a new technique that allows SKETCH to collaborate with a custom solver that handles analysis of similarity functions and synthesizes thresholds while SKETCH makes discrete decisions for the **GBF**. Specifically, SKETCH makes the decisions for (1) expanding the $G_{\mathbf{GBF}}$ grammar with multiple *atoms* or attribute-matching rules, (2) choosing examples in $\mathbf{E}_{\text{SYN}}$ to be positive ($\mathbf{E}_+$) or negative ($\mathbf{E}_-$) for each atom of the expanded **GBF**, (3) choosing the attributes $i \in [1, n]$ and similarity functions $f \in \mathcal{F}$ to be used in these atoms. The custom solver finds a numerical threshold that separates the positive ($\mathbf{E}_+$) and negative examples ($\mathbf{E}_-$) chosen by SKETCH for an atom, if one exists. Otherwise, it will ask the SKETCH solver to backtrack and make alternative discrete decisions. This solver will be called multiple times inside SKETCH. Its pseudocode is given in Algorithm 2.

As an optimization, to avoid recomputing numerical functions in the special purpose solver, we enumerate and memoize the function evaluations on all possible values that can

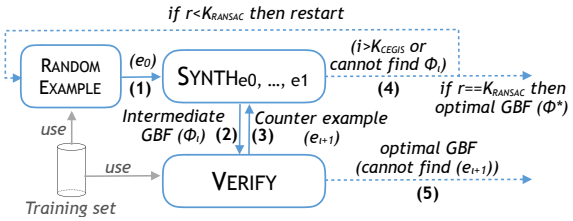


**Figure 2: RuleSynth overview for EM-Synth**

be obtained from aligned attributes in the examples. For example, if we have just one example $e_1 = (r, s)$ with

$$r \equiv \{\text{name} = \text{'C. Zeta-Jones'}, \text{gender} = \text{'F'}\}$$
$$s \equiv \{\text{name} = \text{'Catherine Zeta-Jones'}, \text{sex} = \text{'F'}\}$$

then we evaluate the `Jaccard` similarity function on aligned attributes and provide the following table `evalSimFn` to the custom solver (Algorithm 2):

| example id | matched attribute | function | evaluation |
|---|---|---|---|
| $e_1$ | name\|name | Jaccard | 0.5 |
| $e_1$ | gender\|sex | Jaccard | 1.0 |

**Algorithm 2:** Custom solver inside SKETCH

**input :** $f$ : Chosen similarity function
$a$ : Matched attribute Id
$\mathbf{E}_+$ : Examples chosen to be positive
$\mathbf{E}_-$ : Examples chosen to be negative
**output:** *exists* : Does a valid threshold exist
$\theta$ : A valid threshold separating $\mathbf{E}_+$ & $\mathbf{E}_-$

```
1  θ_atmost ← 1.0
2  for e ∈ E_+ do
3      θ_atmost = min (θ_atmost, evalSimFn(e, a, f))
4  θ_atleast ← 0.0
5  for e ∈ E_− do
6      θ_atleast = max (θ_atleast, evalSimFn(e, a, f))
7  if θ_atleast < θ_atmost then
8      exists ← true
9      θ ← (θ_atleast + θ_atmost) / 2
10 else
11     exists ← false
```

**Synthesis from a few Examples (CEGIS).** We use ideas from the Counter-Example Guided Inductive Synthesis (CEGIS) [38] approach to build an iterative algorithm that has two phases: **Synth** (line 8) and **Verify** (line 11). The idea is to iteratively synthesize a **GBF** that works for a small set of examples and expand this set in a smart manner by adding only those examples that are currently not being handled correctly by the synthesized **GBF**.

For example, consider Figure 1 with matching examples $\mathbf{M} = \{(r_1, s_1), (r_4, s_2), (r_2, s_1)\}$ and non-matching examples $\mathbf{D} = \{(r_1, s_2), (r_4, s_1)\}$. Suppose the algorithm picks $(r_1, s_1)$ as the first example and **Synth** returns the function $\Phi_0 =$ `Equal[name]` $\geqslant 1.0$. **Verify** tries this function on all examples in $\mathbf{M} \cup \mathbf{D}$ and randomly picks $(r_4, s_1)$ as the counter-example, i.e., an example which is not correctly matched by the function $\Phi_0$ since the names are not equal for $(r_4, s_1)$. It would then add this counter-example to the set $\mathbf{E}_{\text{SYN}}$ and start the next CEGIS iteration. In this iteration **Synth** may now return the function $\Phi_1 =$ `Jaccard[name]` $\geqslant 0.4$, which matches all examples correctly.

At iteration $i$, **Synth** uses the currently available examples $\mathbf{E}_{\text{SYN}} = \{e_0, e_1, \ldots, e_i\}$ and solves the Exact SyGuS problem with SKETCH to find a **GBF** $\Phi_i$ from the bounded grammar $G_{\mathbf{GBF}}(N_a, N_d)$ that correctly handles all the examples in $\mathbf{E}_{\text{SYN}}$. **Verify**, on the other hand, considers the

full set of examples $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$ and finds the *counter-example* subset $\overline{\mathbf{E}}_{\Phi_i} \subset \mathbf{E}$, which contains examples $e \in \mathbf{E}$ such that $\Phi_i(e) = \mathit{false}$ if $e \in \mathbf{M}$ and $\Phi_i(e) = \mathit{true}$ if $e \in \mathbf{D}$. In other words, it identifies examples that are incorrectly handled by $\Phi_i$. A counter-example $e_{i+1}$ chosen randomly from $\overline{\mathbf{E}}_{\Phi_i}$ is added to the set $\mathbf{E}_{\text{SYN}}$ to be considered in the next **Synth** phase. The process continues until either **Synth** is unable to find a **GBF** for the current set of examples or until it has performed $K_{\text{CEGIS}}$ (*CEGIS cutoff*) iterations. If **Verify** cannot find any counter-example (i.e., $\overline{\mathbf{E}}_{\Phi_i} = \varnothing$), the algorithm terminates and outputs $\Phi_i$ as the optimal **GBF** since it correctly handles all examples in $\mathbf{E}$.

**Synthesis with Sub-optimal Examples (RANSAC).** We use ideas from the Random Sample Consensus (RANSAC) [20] approach and build a loop on top of the CEGIS loop to restart it multiple times with different initial random examples ($e_0$). The idea is that if the provided example set contains a small number of sub-optimal examples, then multiple runs are more likely to avoid them. Note that some examples individually may not be sub-optimal, i.e., the algorithm may still find a good **GBF** after choosing them in the CEGIS loop. Instead, certain larger subsets of examples may correspond to conflicting constraints on the **GBF** grammar and constitute sub-optimality only when all examples in that subset are chosen together. Both the randomness in **sample** routine and the RANSAC restarts help avoid choosing all such points together. Before restarting CEGIS, if the number of restarts reaches $K_{\text{RANSAC}}$ (the RANSAC cutoff) then the algorithm terminates and outputs the *best* **GBF** $\Phi^*$ seen across all CEGIS and RANSAC iterations *w.r.t.* the optimality metric $\mu$.

# 5. SYNTHESIS OPTIMIZATIONS

## 5.1 Grammar: Conciseness and Null Values

We use the power of synthesis to control the structure of the **GBF** and provide a concise formula as the output. Note that these techniques also help us avoid over-fitting to the provided examples since our **GBF**s are as small as possible.

**Handling Null values in $G_{\mathbf{GBF}}$:** Null (missing) values are problematic because we cannot know whether two records match on some attribute $A$ if one record has a Null value for $A$. Rather than assuming that such records do not match (as was done in previous work), we learn different rules for the Null and noNull case. We specify a new grammar production in $G_{\mathbf{GBF}}$ for deriving **GBF**s that capture this intuition:

$$\begin{array}{lll} \textsf{grammar} & G_{\mathbf{GBF}} \rightarrow & \textbf{if } (\approx(i, \texttt{noNulls}, 1.0)) \\ & & \textbf{then } (G_{\mathbf{GBF}}) \textbf{ else } (G_{\mathbf{GBF}}) \\ & & i \in [1, n] \end{array}$$

It says that if there are no nulls in the matching attributes in a pair of records, then we should use one **GBF**; otherwise we should use a different **GBF**. This makes it possible for the synthesizer to quickly find rules similar to example $\varphi_4$ (Section 1). Note that this addition does not affect the expressibility of the grammar and is purely for making the grammar $G_{\mathbf{GBF}}$ and the synthesis process more targeted towards databases with large numbers of nulls.

**Incremental Grammar Bounds** To make sure that the generated rules are small and concise, RuleSynth iteratively adjusts the grammar bound on the number of attribute-matching rules ($N_a$) as it runs, starting with rules of size 1 and growing up to $N_a$, so that it prefers smaller rules when they can be found. To be more precise, we introduce the following loop in Algorithm 1 replacing line 8:

**Procedure** Incremental Grammar Bounds

```
1 n_a ← 1                    // attribute-matching rules bound n_a
2 while n_a ⩽ N_a do
3 |   Φ_i ← Synth(G_DNF(n_a, N_d), E_SYN, F)
4 |   if Φ_i = null then                  // Unsatisfiable Synth
5 |   |   n_a ← n_a + 1                        // try larger n_a
6 |   else
7 |   |   break
```

RuleSynth uses an optimized version of this loop where in CEGIS iteration $i \geqslant 1$, the initial value of $n_a$ is set to the value of $n_a$ used to synthesize $\Phi_{i-1}$ in the previous CEGIS iteration (instead of starting with $n_a = 1$). Since the set of examples being considered in iteration $i$ is a superset of examples considered in iteration $i-1$, if for any $n_a$ **Synth** could not find a **GBF** in iteration $i-1$ then for the same $n_a$ it will not be able to find a **GBF** that matches all the examples in iteration $i$.

## 5.2 Sampling: Bias in Picking Examples

In CEGIS iteration $i$, the RuleSynth algorithm tries to primarily choose an example that is currently not being matched correctly. This guides the resulting **GBF** towards higher accuracy on the example set by making more and more examples match correctly. On top of this, RuleSynth picks the best **GBF** that maximizes $\mu$ across all CEGIS and RANSAC iterations. For optimality metrics like $\mu_{\text{F-measure}}, \mu_{\text{precision}}, \mu_{\text{recall}}$ it is important to focus on finding **GBF**s that maximize the number of positive examples being matched correctly. Note that if the set of examples is largely only negative examples then the likelihood of most of the chosen examples being negative is high. This may result in the algorithm missing certain positive examples for smaller CEGIS cutoffs ($K_{\text{CEGIS}}$) and thereby finding a solution with possibly lower $\mu$ even when the accuracy is high. Hence, in RuleSynth we eliminate this bias based on the actual distribution of positive and negative examples and replace it with a 50-50 chance of choosing a positive or negative example, i.e., the **sample** routine (line 5 in Algorithm 1) is modified as described above.

## 5.3 Algorithm Optimizations

**SynthComp: Composition of Discovered Rules.** The SynthComp optimization efficiently produces larger **GBF**s from smaller **GBF**s. SynthComp supplements RuleSynth with an additional step at the end, where it synthesizes a general boolean formula (**GBF**) using $B_{\text{SynthComp}}$ **GBF**s from a set of top $K_{\text{SynthComp}}$ **GBF**s collected across all CEGIS and RANSAC iterations while maximizing the metric $\mu$. That is, if RuleSynth found 3 functions $\varphi_1, \varphi_2, \varphi_3$ with metric $\mu$ being $0.82, 0.77, 0.64$, respectively, then it will look at all boolean combinations of these **GBF**s and come up with the best one, say, $(\varphi_1 \wedge \varphi_3) \vee \varphi_2$ with metric $\mu = 0.87$ that is better than all three functions individually. Our implementation of SynthComp uses ideas from truth-table based synthesis of Boolean circuits [15] and takes less than 5 minutes for $B_{\text{SynthComp}} \leqslant 4$ and $K_{\text{SynthComp}} \leqslant 10$. Note that a larger value of $B_{\text{SynthComp}}$ would make this time grow substantially, but it will also make the final rules large and uninterpretable. As will be seen in the experiments

(Section 6), these values ($K \leqslant 10, B \leqslant 4$) work well in RS-SynthComp method for all datasets and lead to effective and interpretable rules, which are generated in reasonable time. Using SynthComp also involves a tradeoff between conciseness (number of attribute-matching rules in **GBF**) and performance (metric $\mu$).

**Consensus: Building Consensus of Multiple Rules.** Consensus optimization (similar to SynthComp) builds a combination of discovered rules or **GBF**s. But, unlike SynthComp, Consensus focuses on combinations of rules of a specific form. More specifically, Consensus optimization searches for rules of the form:

$$\texttt{count\_true}\,(\varphi_1, \varphi_2, \ldots, \varphi_B) \geqslant C$$

where $\varphi_1, \varphi_2, \ldots, \varphi_B$ are $B$ **GBF**s discovered by RuleSynth, `count_true` represents the function that counts how many of these rules output `true` when evaluated on an example and $C$ is an integer between 0 and $B$. Intuitively, the Consensus optimization finds a rule that builds a consensus of at least $C$ out of $B$ rules when classifying an example.

To find such a combination, the Consensus optimization enumerates all $B$-combinations of top $K$ **GBF**s and then tries different values of $C$ (from 0 to $B$) and evaluate the metric $\mu$ on all of them. Afterwards, it picks the best consensus rule found so far. Note that since we are considering only some of the all possible compositions of discovered rules, the search space here is much smaller than SynthComp and hence, we can run this procedure with larger bounds in the same amount of time. Our implementation of Consensus takes less than 10 minutes for $B \leqslant 5$ and $K \leqslant 15$. As will be seen in the experiments (Section 6), these values lead to more effective rules. We will use the notation $B_{\text{Consensus}}$ and $K_{\text{Consensus}}$ to distinguish these parameters from those of SynthComp.

# 6. EXPERIMENTS

The key questions we answer with our evaluation are: (i) How do our rules compare in interpretability and accuracy to other interpretable models? (Exp-1, Exp-2); (ii) How do they compare in accuracy to expert-provided rules? (Exp-3); (iii) How do they compare in accuracy to non-interpretable models, such as SVMs? (Exp-4); (iv) How do we perform when using limited training data? (Exp-5); (v) Can RuleSynth discover rules in reasonable amounts of time? (Exp-6); and (vi) How efficient are the RuleSynth rules compared to non-interpretable ML models when applied to large datasets? (Exp-7).

## 6.1 Experimental Setup

**Datasets.** Table 1 shows four real-world datasets used in our evaluation. The Cora dataset has one relation, while the others have two relations with aligned schemas. Positive examples for every dataset are also given. To ensure that negative examples are quite different from each other, we took the Cartesian product of the relations and pruned pairs with high Jaccard tri-gram similarity values [24, 25]. We varied the similarity threshold across datasets to control the number of negative examples.

Table 1 also shows the average number of record pairs with at least one null value. These numbers show the importance of using the custom noNulls function in a formula because noNulls in the `if` condition enables the synthesizer to find smaller rules for the noNulls (`then`) vs nulls (`else`) cases. Some datasets have a skewed distribution of nulls across attributes, e.g., for **DBLP-Scholar**, the attribute year has around $40K$ nulls, whereas title and authors have 0.

| | #Matching Pairs | #Record Pairs | #Attr | Avg #nulls per Attr |
|---|---|---|---|---|
| $\mathsf{D_C}$ | 14,280 | 184,659 | 9 | 92,955(50%) |
| $\mathsf{D_{AG}}$ | 1,300 | 97,007 | 4 | 22,583(23%) |
| $\mathsf{D_{LF}}$ | 6,048 | 341,244 | 10 | 99,629(29%) |
| $\mathsf{D_{DS}}$ | 5,347 | 112,839 | 4 | 12,685(11%) |

$\mathsf{D_C}$ =**Cora**, $\mathsf{D_{DS}}$ =**DBLP-Scholar**
$\mathsf{D_{AG}}$ =**Amazon-GoogleProducts**, $\mathsf{D_{LF}}$ =**Locu-FourSquare**

**Table 1: Dataset statistics**

**Inputs for EM-GBF Problem (Section 2.2).** In the following, we use F-measure as the metric to be optimized. We use a set of 29 similarity functions that were also used in the SIFI project [40]. This set includes functions from the Simmetrics library (`https://github.com/Simmetrics/simmetrics`) and functions implemented by authors of SIFI. We also treat Equal and `noNulls` as two similarity functions that evaluate to 0 or 1. We use the outputs of these similarity functions rounded to a finite precision of 3 decimals.

**Input features for ML techniques.** For every example record pair, we evaluate all available similarity functions on strings from aligned attributes and construct a vector of these numerical values between 0 and 1. These vectors are used as input feature vectors for all ML techniques. For SVM, we also normalize the feature vectors to have zero mean and unit variance during training and use the same scaling while testing [3].

**Comparisons with State-of-the-Art ML Approaches.** We compared the basic and the optimized variants of RuleSynth with decision trees, SVM [40], gradient tree boosting [9] and random forests [22]. All ML methods convert EM into a binary classification problem.

While the output from SVM lacks logical interpretability, a decision tree can be interpreted as a boolean formula with multiple **DNF** clauses arising from traversal of paths that lead to positive classification. However, the output of Random Forests is tedious to interpret because: (1) the output has tens to hundreds of trees that are aggregated to make the final decision, (2) each decision tree has a large depth resulting into thousands of nodes, making them hard to interpret individually. Since a similar aggregation mechanism, such as bagging, can be used over RuleSynth as well, we focus on (2) and compare our results with a single decision tree from [22].

**Comparisons with Rule-based Learning Approaches.** We evaluated RuleSynth against a heuristic-based approach, SIFI [40], which searches for optimal similarity functions and thresholds for the attribute comparison given a DNF grammar provided by a human expert. In contrast, the **GBF**s are automatically discovered by RuleSynth without any expert-provided structure of the rules.

**Implementation.** All experiments were run on a machine with Ubuntu 14.04 OS, 32 GB RAM and 16-core 2.3 GHz CPU. We implemented RuleSynth in Python 2.7.6 as scripts that interact with the Sketch synthesis tool (written in Java and C++). We implemented SVM in Python using the scikit library and libsvm (`http://www.csie.ntu.edu.tw/~cjlin/libsvm`). The two baseline approaches, i.e., SIFI and decision trees, were obtained from the authors of [40]

and [22], respectively. SIFI was implemented in C++, and the random forest entity matcher was coded in Java using the Weka library [36].

**Techniques and Parameters.** For all ML techniques, we used a simple grid search [4] of values for different parameters. We list the parameters being searched for below: (i) For decision trees: depth of the tree, minimum number of examples needed for a split. (ii) For SVM: choice of kernel (LinearSVC or RBF) [2], the penalty hyper-parameter $C$ in the loss function and $\gamma$ hyper-parameter for RBF kernel. (iii) For Gradient Boosting: the learning rate, maximum depth of a tree, maximum number of trees. (iv) For Random Forests: maximum depth of a tree, number of trees.

For decision trees, we separately present results for depths $3, 4$ and $10$ (the default configuration in Weka). For SVM, we separate the results for the two kernels. For Gradient Boosting and Random Forests, we present results with *small* #-atoms, i.e., $2-4$ trees of depth $2-4$ (so that #-atoms is bounded by 60), and *large* #-atoms (searching around the defaults in the Scikit learn [31] libraries on the grid), i.e., $5-15$ depth or unlimited depth trees for Random Forests and $25-100$ trees with depth $2-4$ for Gradient Boosting. Note that, even though these two techniques have interpretable trees, each tree or leaf has a numerical weight assigned to it that makes them hard to interpret. For SVM we use balanced class-weights as a low-effort configuration for optimizing F-measure [29]. We also ran SIFI with default configurations and grammars given by experts.

We use three variants of our algorithm: (1) the basic CEGIS+RANSAC based RuleSynth (Section 4.2), (2) RS-SynthComp that uses the SynthComp optimization (Section 5.3), and (3) RS-Consensus that uses the Consensus optimization (Section 5.3). For all variants, we have the following parameters with their respective default values: (i) The depth of the grammar $N_d = 4$, which is enough to represent formulas with at most 15 atoms; (ii) A high $K_{\text{CEGIS}} = 1000$ with a timeout of 15 minutes per CEGIS iteration so that the CEGIS loop runs until it finds a set of examples for which Sketch cannot synthesize a valid rule or it times out and picks the best rule obtained till then; (iii) The bound $K_{\text{RANSAC}} = 5$ to restart CEGIS 5 times and explore different underlying sets of examples; (iv) The number of attribute-matching rules $N_a$: for RuleSynth, we set $N_a = 8$ so that it is comparable with the number of atoms in a decision tree of depth 3. For RS-SynthComp we use $N_a = 5$ and combine $B_{\text{SynthComp}} = 3$ rules out of $K_{\text{SynthComp}} = 10$ rules (Section 5.3) to generate a composite **GBF** so that in total #-atoms is bounded by 15 and is comparable with #-atoms in a decision tree of depth 4. For RS-Consensus, we use $N_a = 8$ and combine $B_{\text{Consensus}} = 5$ rules out of $K_{\text{Consensus}} = 15$ rules (Section 5.3) to have similar #-atoms as the *small* Gradient Boosting Trees and Random Forests (with $2-4$ trees of depth $2-4$).

**Performance Evaluation.** We performed $K$-fold cross-validation (for $K$=5) on each of the datasets used, where we divided the data into $K$ equal fractions (folds) randomly and performed $K$ experiments. In each experiment one of the $K$ folds was the test set while the remaining $K-1$ folds were training. We report the average F-measure obtained across all folds on the test sets as the performance metric (Figure 5). Note that we use the same folds for each technique we compare and for each fold we may find different optimal values for the parameters of the ML techniques.

## 6.2 Experimental Results

**Exp-1: Interpretability.** We measure interpretability as being inversely proportional to the number of *attribute-matching rules* (or atoms) present in the rule. In other words, interpretability is defined as the number of atomic similarity function comparisons with a threshold $\approx(i, f, \theta)$ in the formula representing the rule. For clarity, we represent atoms or attribute-matching rules as $\big(fn[attr] \geqslant \theta\big)$, where $fn$ is the name of the applied similarity function, $attr$ is the name of the matched attribute, and $\theta$ is the corresponding threshold, e.g., EditDistance[title] $\geqslant 0.73$ is a valid atom. Intuitively, a complex **DNF** is less interpretable than a semantically equivalent but more concise **GBF**.

Below, we present two **GBF**s, $\varphi_{synth}$ and $\varphi_{tree}$ for Cora, obtained by using RuleSynth and decision trees of depth 3, respectively. We obtained both **GBF**s on the same training set as the best rules. These rules result in average F-measures of 0.83 ($\varphi_{synth}$) and 0.77 ($\varphi_{tree}$) on test data. The **GBF** $\varphi_{synth}$ demonstrates the conciseness of formulas generated by RuleSynth as compared to $\varphi_{tree}$, as $\varphi_{synth}$ has only 6 atoms whereas $\varphi_{tree}$ has 12 atoms. Also note that the RuleSynth rules include if/then/else clauses that allow them to be more compact that the DNF-based rules the decision tree produces.

$\varphi_{synth}$ :
( ChapmanMatchingSoundex[author] $\geqslant 0.937$
  $\bigwedge$ **if** noNulls[date] $\geqslant 1$
  **then** CosineGram$_2$[date] $\geqslant 0.681$
  **else** NeedlemanWunch[title] $\geqslant 0.733$) $\bigvee$
( EditDistance[title] $\geqslant 0.73$
  $\bigwedge$ OverlapToken[venue] $\geqslant 0.268$)

$\varphi_{tree}$ :
( OverlapGram$_3$[title] $\geqslant 0.484$
  $\bigwedge$ MongeElkan[volume] $\geqslant 0.429$
  $\bigwedge$ Soundex[title] $\geqslant 0.939$) $\bigvee$
( OverlapGram$_2$[pages] $\geqslant 0.626$
  $\bigwedge$ MongeElkan[volume] $\geqslant 0.429$
  $\bigwedge \neg$ (Soundex[title] $\geqslant 0.939$)) $\bigvee$
( ChapmanMeanLength[title] $\geqslant 0.978$
  $\bigwedge \neg$ (OverlapGram$_3$[author] $\geqslant 0.411$)
  $\bigwedge \neg$ (MongeElkan[volume] $\geqslant 0.429$)) $\bigvee$
( CosineGram$_2$[title] $\geqslant 0.730$
  $\bigwedge$ OverlapGram$_3$[author] $\geqslant 0.411$
  $\bigwedge \neg$ (MongeElkan[volume] $\geqslant 0.429$))

Figure 3 shows the interpretability results with respect to the number of atoms for all datasets. It shows that our algorithm produces more interpretable rules, i.e., with fewer atoms, than decision trees with depths 3 and 4 for all datasets. In particular, RuleSynth produces rules that are (i) more interpretable than decision trees with depth 3 for all datasets and (ii) up to eight times more interpretable than decision trees with depth 4 (see dataset `Amazon-GoogleProducts`). The second variants RS-SynthComp produces rules with more atoms but still has better interpretability than decision trees with depth 4. Moreover, as we will see in Exp-2, the rules produced by RS-SynthComp are more effective than decision trees with both depth 3 and 4. The third variant RS-Consensus produces rules with even more atoms but they still have less #-atoms than *small* Gradient Boosting and Random Forests. Small Gradient Boosting and Random Forests are also not easily interpretable due to the presence of numerical weights along with the trees.

Figure 3 also tells us that the number of atoms increases exponentially with the depth of the decision trees i.e., the deeper is the tree, the less interpretable the corresponding

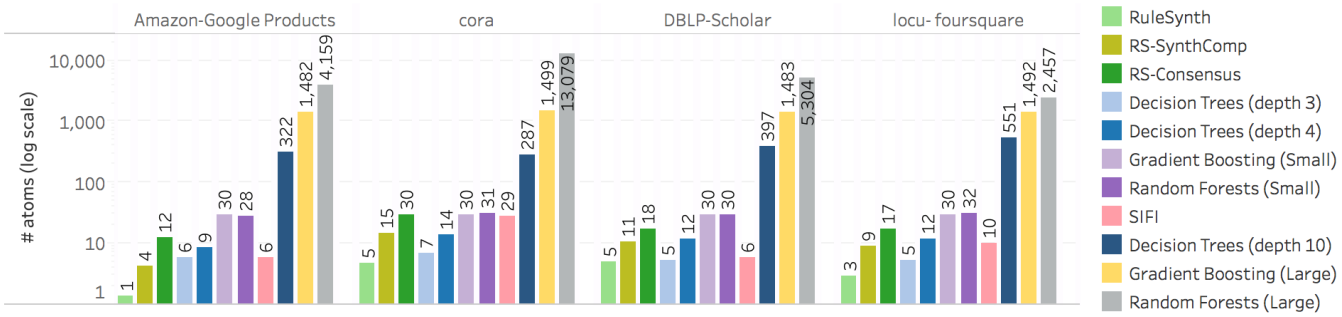


**Figure 3: Interpretability results for 5-folds experiment (80% training and 20% testing data)**

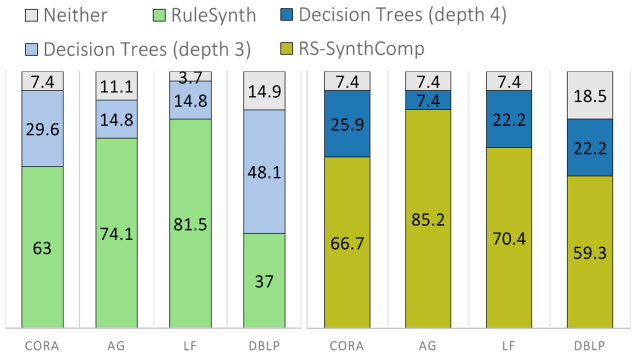


**Figure 4: User interpretability preference: Cora, Amazon-GoogleProducts (AGP), Locu-FourSquare (LFS), DBLP-Scholar (DBLP)**

rules are. For example, it is nearly impossible to interpret decision trees of depth 10 with thousands of atoms.

*User study.* Figure 4 shows the results of our informal user study with 27 CS researchers from six institutions. We gave each participant 8 multiple-choice questions with 3 options for the answer. Each question comprised a pair of well-formatted rules generated by two different techniques from the same training data. The participant was asked to select *which one of the two rules they thought was more interpretable*. Each participant was given 2 questions for each dataset. One question compared the rules generated by RuleSynth against decision trees of depth 3, and the other compared RS-SynthComp against decision trees of depth 4. We observe from the results that the rules with fewer atoms are preferred by more users. Generally the rules generated by RuleSynth and its variants are preferred except in one case (i.e., RuleSynth on `DBLP-Scholar`), where the decision trees have a similar number of atoms as our algorithms, as shown in Figure 3. On average, 67.15% of the responses state that the rules generated by our algorithms are more interpretable, while only 23.13% prefer the decision trees, and 9.72% state no preference. This supports the validity of #-atoms as our measure of interpretability.

These results overall showcase the power of program synthesis to generate concise formulas from a rich grammar, yielding compact and more interpretable rules.

**Exp-2: Effectiveness vs. Interpretable Decision Trees.** We now evaluate the effectiveness of rules generated by our algorithms against the ones found by decision trees. As mentioned before, we use the average F-measure across 5 folds as the effectiveness metric.

Figure 5 shows the average F-measures for different interpretable techniques. We observe that RuleSynth achieves a higher F-measure than decision trees with depth 3 for all datasets, except for `DBLP-Scholar` where the F-measures are comparable. Decision trees achieve higher F-measures when increasing their depth from 3 to 4 for all datasets. However, RS-SynthComp still results into higher F-measures than decision trees with depth 4 on all data sets.

Moreover, as we saw in Figure 3, each of RuleSynth and RS-SynthComp produces more interpretable rules than decision trees with depth 4 for all datasets. From figures 3 and 5, we conclude that decision trees can get better F-measures by increasing their depth but this comes at a significant sacrifice to their interpretability. In contrast, RuleSynth can get better F-measures by applying SynthComp and Consensus optimizations while not sacrificing on interpretability as much. For example, for the `Amazon-GoogleProducts` dataset, increasing the depth of decision tree from 3 to 4 increases the F-measure from 0.484 to 0.553 while the average number of atoms increases from 4.6 to 10.2. In contrast, the SynthComp optimization increases the F-measure from 0.567 to 0.614 while increasing the average number of atoms from 1.4 to 4.2.

**Exp-3: Effectiveness vs. Expert-Provided Rules.** To further demonstrate the effectiveness of **GBF**s produced by RuleSynth and its variants, we compare RuleSynth with SIFI [40]. SIFI requires experts to provide a **DNF** template from experts as an input and completes it to generate a rule. In contrast, RuleSynth discovers rules automatically, reducing the effort needed from an expert.

Figure 5 shows that RuleSynth and its variants perform better than SIFI for all datasets. In contrast with SIFI, which employs a heuristic to search through a smaller space of rules, RuleSynth searches through a huge space of generic **GBF**s. This allows us to discover various corner cases that can be sometimes missed by an expert-provided expression. In addition, as shown in Figure 3, RuleSynth generates **GBF**s that are more concise (and thus interpretable) than the **DNF**s produced by SIFI for all datasets.

**Exp-4: Effectiveness vs. Non-interpretable Methods.** We now compare RuleSynth and its variants with four ML algorithms: (1) Decision trees with depth 10, (2) SVM, (3) Random Forests, and (4) Gradient Tree Boosting. Figure 5 shows the results for interpretable methods, and Figure 6 gives the results for non-interpretable methods, we observe that all the three variants of RuleSynth achieve smaller F-measure values than the ML algorithms on an average. Still, RS-Consensus achieves quite comparable F-measures, with the F-measure difference between the ML best algorithm and RS-Consensus being 0.08, 0.05, 0.02, and 0.04 for each of the four data sets. However, the

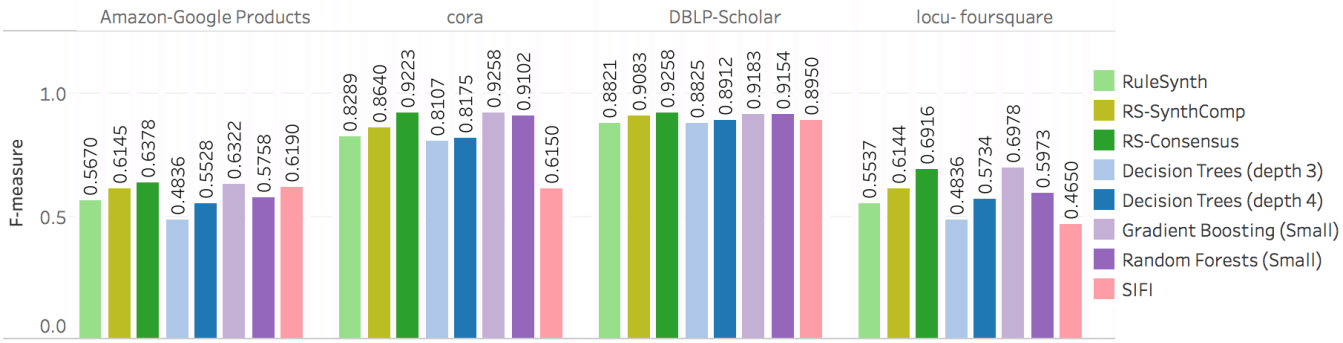


**Figure 5: Effectiveness results for 5-folds experiment (80% training and 20% testing data)**

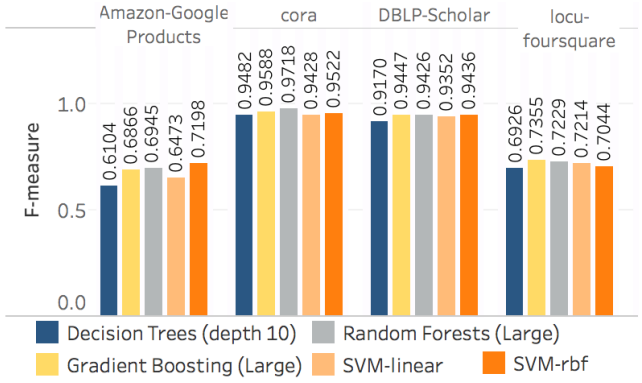


**Figure 6: Effectiveness results for 5-folds experiment: non-interpretable methods**

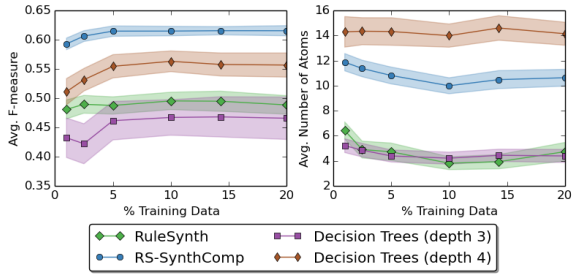


**Figure 7: Locu-Foursquare (100 runs with 99% confidence intervals on the means in the shaded regions)**

effectiveness of these ML algorithms comes at a high price. We see in Figure 3 that these four ML algorithms are not interpretable: (i) SVM does not produce rules, (ii) decision trees with depth 10 yield rules with around $1K$ atoms for all datasets, (iii) Random Forests and Gradient Boosting provide both rules with 1K-13K atoms with hundreds of weights, which are also impossible to interpret for a human.

**Exp-5: Variable Training Data.** We also varied the number of folds ($K$) by randomly sampling a fraction $\frac{1}{K}$ of training examples with $K = 100, 40, 20, 10, 7, 5$. Each fraction $\frac{1}{K}$ corresponds to a different percentage $P\%$ of examples (i.e., $P = 1, 2.5, 5, 10, 14.3, 20$). We use the rest $(100 - P)\%$ of the examples for testing, and we train and test on 100 such randomly selected sets for each percentage $P$. We report the average test-set F-measure and size of matching rules obtained across all 100 runs (99% confidence intervals).

Figure 7 shows the comparison between interpretable decision trees and RuleSynth variants on Locu-Foursquare dataset with different percentages (1% to 20%) of training data. The figures for the other datasets show similar trends, and are thus omitted for space limitation. We compare Decision Trees (depth 3) with RuleSynth since they both produce rules with smaller sizes, and, Decision Trees (depth 4) with RS-SynthComp since they both produce interpretable rules with larger sizes. Both RuleSynth and RS-SynthComp outperform Decision Trees of depth 3 and 4, respectively, in effectiveness (higher F-measure) on all datasets. At the same time, RS-SynthComp generates more interpretable (lower number of atoms) rules than Decision Trees (depth 4). RuleSynth and Decision Trees (depth 3) both generate small and interpretable rules (2-7 atoms on average). RuleSynth generates smaller rules for 2 out of 4 datasets, has similar interpretability for `Locu-Foursquare`, and, generates slightly larger rules for `DBLP-Scholar`. RS-SynthComp is the most effective method for generating interpretable rules (2-14 atoms on average) with limited training data on all datasets.

**Exp-6: Efficiency of Training.** RuleSynth and its variants provide the flexibility for users to control how much the algorithm should explore in the CEGIS loop (bound $K_{\text{CEGIS}}$, time-limit, grammar bounds) and how many times it should restart (bound $K_{\text{RANSAC}}$).

Figure 8 shows that RuleSynth and its variants take at most an hour to search through the huge space of rules in order to produce an effective and concise rule as output for all datasets in Table 1. This is a reasonable amount of time as compared to what it takes experts to examine the dataset and write their own rule expressions, especially given the low-cost of computation relative to human time. For example, our experts took around 2 hours on average to write a **DNF** expression for SIFI per dataset.

Figure 8 also shows that SIFI searches through a smaller constrained space in at most 40 minutes to produce a rule. Decision trees with depth 3 and 4 produce a rule in less than a minute but the produced rules are neither as concise nor as effective as rules produced by RuleSynth and its variants (Exp-2). Both decision trees with depth 10 and SVM take at most 8 minutes to produce a rule but they are not designed to expose interpretable results (Exp-4).

**Exp-7: Efficiency of Testing.** When we have $N$ records and we want to apply a rule or a classifier on each pair of these records to identify duplicates. In general, applying a rule would require enumerating all $O(N^2)$ pairs and computing the relevant similarity functions. Note that until now, for training, we pre-computed these similarity functions but, now, for testing the rule in a new environment, we have to compute the relevant similarity functions again to be able to apply the rule or the classifier. For a classifier that uses many similarity functions, this process becomes

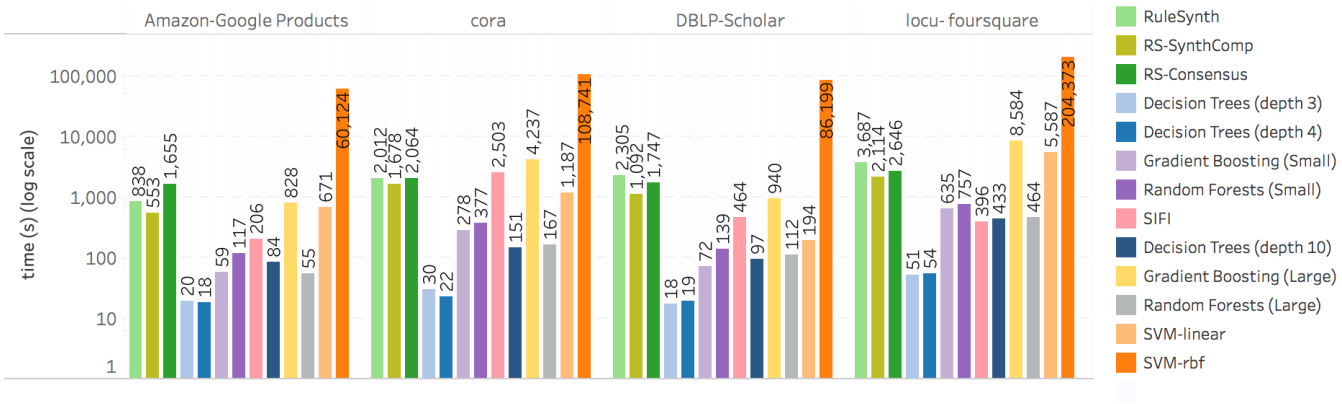


**Figure 8: Efficiency (average time for training per fold) for 5-folds experiment (80% training / 20% testing)**

| | rule | # record pairs | time taken (s) | | |
|---|---|---|---|---|---|
| | | | SVM | rule on all pairs | rule on buckets |
| $D_C$ | $\varphi_1$ | 360,000 | 3,576.18 | 55.17 | 2.56 |
| | $\varphi_2$ | | | 32.05 | 2.67 |
| | $\varphi_3$ | | | 87.91 | 3.63 |
| | $\varphi_1$ | 810,000 | N/A | 127.29 | 8.30 |
| | $\varphi_2$ | | | 61.38 | 5.54 |
| | $\varphi_3$ | | | 176.46 | 8.85 |
| $D_{AG}$ | $\varphi'_1$ | 360,000 | 12,528.3 | 30.28 | 0.80 |
| | $\varphi'_2$ | | | 34.47 | 1.02 |
| | $\varphi'_3$ | | | 98.94 | 1.17 |
| | $\varphi'_1$ | 810,000 | N/A | 71.80 | 1.03 |
| | $\varphi'_2$ | | | 87.23 | 1.9 |
| | $\varphi'_3$ | | | 216.61 | 3.65 |
| $D_C$ =**Cora**, $D_{AG}$ =**Amazon-GoogleProducts** | | | | | |

**Table 2: Efficiency on testing**

prohibitively slow since they have to compute all of them; as shown in Table 2, SVM is much slower than rules applied on all pairs. Hence, applying smaller rules on all pairs already has an advantage over large classifiers that utilize many similarity functions with numerical weights like SVM. Moreover, for smaller **GBF** rules with a specific structure, one can use a hashing scheme [41] to bucket similar records as a first pass, which reduces the pairwise comparisons from $O(N^2)$ to $O(M^2)$ where $M \ll N$. The rule application on buckets takes much less time ($1-4$s) than applying the rule on all pairs ($30-200$s), as shown in Table 2.

We identified 3 RuleSynth generated rules each for two datasets that are of the form $(\mathsf{f}_{\mathsf{sim}}[\mathsf{attr}] \geqslant)\,\theta \wedge \varphi'$ where $\varphi'$ is a general **GBF** and $f_{sim}$ is a similarity function for which there is a locality sensitive hashing (LSH) family available [41]. Out of 30 functions considered in this work, LSH families are available for at least 7 of them. Note that in RuleSynth, we can also force this structure for all rules with our flexible grammar. For this experiment, we found rules generated by RuleSynth that have the format mentioned above with $f_{sim}$ being the Jaccard function over the set of $n$-grams (with $n = 2$) of the input strings. Using a MinHash LSH scheme described in [28] and available as the Datasketch tool [1], we built an index on the attribute attr with Jaccard threshold $\theta$ to identify potentially similar pairs and reduce the number of record pairs to compare. The running times for this experiment are shown in Table 2.

## 7. RELATED WORK

*Machine Learning-Based Entity Matching.* Most current solutions are variants of the Fellegi-Sunter model [18], where entity matching is treated as a *classification* problem. Such approaches include SVM based methods [7], decision tree based solutions [22], clustering based techniques [11], and Markov logic based models [35].

As remarked earlier, the main obstacles to deploy them is that humans do not understand and trust them [26], due to the lack of interpretability [27].

*Rule-Based Entity Matching.* Declarative EM rules are normally desirable to end users. Such rules are also popular in the database community since they provide great opportunities for improving the performance at execution time, such as those studied in [12, 16]. However, these approaches typically assume that EM rules are given by domain experts, which, in practice, it is a hard problem.

Closer to our work is [40], which discovers similarity functions and their associated thresholds by assuming that the rule structure is given as a $\mathbf{DNF_{EM}}$. In contrast to it, our approach can discover more expressive **GBF** rules, without users to provide the rule structure.

*Active Learning and Crowdsourcing.* Since good and sufficient training dataset is always hard to get in practice, a natural line of study is how to actively involve users in verifying ambiguous tuple pairs, *a.k.a.* active learning in EM [22, 32]. Also, due to the popularity of crowdsourcing platforms, there have also been efforts of leveraging crowd workers for entity matching problems [19, 22, 39].

We assume that training examples are given so we can learn rules. Active learning can be used to collect examples, which is orthogonal to our study.

*Custom Constraint Solvers.* Using a custom solver inside Sketch is similar to using a special purpose theory solver [21] inside a different class of solvers, namely, the SMT solvers. In the context of program synthesis, we are the first to show how a custom solver can be used to solve synthesis problems efficiently inside a general purpose solver like Sketch.

## 8. CONCLUSION

We presented how to synthesize EM rules from positive-negative examples. Given a high level specification of rules and some examples, our solution uses program synthesis to automatically synthesize the **GBF** rules for EM. We have also presented optimizations based on RANSAC and CEGIS to improve the effectiveness of the optimizer. We showed with extensive experiments that our solution produces rules that are both concise and easy to interpret for end-users, while matching test examples with accuracies that are comparable with the state-of-the-art solutions, despite the fact that those solutions produce non-interpretable results.

## 9. REFERENCES


[1] Datasketch: Minhash lsh. https://ekzhu.github.io/datasketch/lsh.html.

[2] Scikit learn: Support vector machines in practice. http://scikit-learn.org/stable/modules/svm.html.

[3] Standardization, or mean removal and variance scaling. http://scikit-learn.org/stable/modules/preprocessing.html.

[4] Tuning the hyper-parameters of an estimator. http://scikit-learn.org/stable/modules/grid_search.html.

[5] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. Detecting data errors: Where are we and what needs to be done? *PVLDB*, 9(12):993–1004, 2016.

[6] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*. 2015.

[7] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD*, 2003.

[8] P. S. G. C., C. Sun, K. G. K., H. Zhang, F. Yang, N. Rampalli, S. Prasad, E. Arcaute, G. Krishnan, R. Deep, V. Raghavendra, and A. Doan. Why big data industrial systems need rules and what we can do about it. In *SIGMOD*, 2015.

[9] T. Chen. Introduction to boosted trees. *University of Washing Computer Science. University of Washington*, 22, 2014.

[10] L. Chiticariu, Y. Li, and F. R. Reiss. Rule-based information extraction is dead! long live rule-based information extraction systems! In *EMNLP*, pages 827–832, 2013.

[11] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *KDD*, 2002.

[12] N. Dalvi, V. Rastogi, A. Dasgupta, A. Das Sarma, and T. Sarlós. Optimal hashing schemes for entity matching. In *WWW*, 2013.

[13] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The data civilizer system. In *CIDR*, 2017.

[14] H. H. Do and E. Rahm. COMA - A system for flexible combination of schema matching approaches. In *VLDB*, 2002.

[15] R. Drechsler and R. Wille. From truth tables to programming languages: Progress in the design of reversible circuits. In *2011 41st IEEE International Symposium on Multiple-Valued Logic*, 2011.

[16] A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, J. Quiané-Ruiz, N. Tang, and S. Yin. NADEEF/ER: generic and interactive entity resolution. In *SIGMOD*, 2014.

[17] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[18] I. Fellegi and A. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64 (328), 1969.

[19] D. Firmani, B. Saha, and D. Srivastava. Online entity resolution using an oracle. *PVLDB*, 2016.

[20] M. A. Fischler and R. C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6), June 1981.

[21] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Dpll (t): Fast decision procedures. In *CAV*, volume 4, pages 175–188. Springer, 2004.

[22] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014.

[23] Interpretable ML for Complex Systems NIPS 2016 Workshop. `https://sites.google.com/site/nips2016interpretml`.

[24] H. Köpcke and E. Rahm. Training selection for tuning entity matching. In *International Workshop on Quality in Databases and Management of Uncertain Data*, 2008.

[25] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 2010.

[26] H. Lakkaraju, S. H. Bach, and J. Leskovec. Interpretable decision sets: A joint framework for description and prediction. In *KDD*, 2016.

[27] T. Lei, R. Barzilay, and T. S. Jaakkola. Rationalizing neural predictions. In *EMNLP*, 2016.

[28] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[29] D. R. Musicant, V. Kumar, and A. Ozgur. Optimizing f-measure with support vector machines. In *Proc. of the 16$^{th}$ International Florida Artificial Intelligence Research Society Conference*, 2003.

[30] F. Panahi, W. Wu, A. Doan, and J. F. Naughton. Towards interactive debugging of rule-based entity matching. In *EDBT*, pages 354–365, 2017.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[32] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *KDD*, 2002.

[33] L. Seligman, P. Mork, A. Y. Halevy, K. P. Smith, M. J. Carey, K. Chen, C. Wolf, J. Madhavan, A. Kannan, and D. Burdick. Openii: an open source information integration toolkit. In *SIGMOD*, 2010.

[34] R. Singh, V. Meduri, A. K. Elmagarmid, S. Madden, P. Papotti, J. Quiané-Ruiz, A. Solar-Lezama, and N. Tang. Generating concise entity matching rules. In *SIGMOD*, pages 1635–1638, 2017.

[35] P. Singla and P. Domingos. Entity resolution with markov logic. In *ICDM*, 2006.

[36] T. C. Smith and E. Frank. *Statistical Genomics: Methods and Protocols*, chapter Introducing Machine

Learning Concepts with WEKA. Springer, 2016.

[37] A. Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, 2009.

[38] A. Solar-Lezama. Program sketching. *STTT*, 15(5-6), 2013.

[39] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 2012.

[40] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *PVLDB*, 2011.

[41] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *CoRR*, 2014.